

# 智能推荐系统第一次作业： 基于协同过滤的用户评分预测

张宏伟  
10205102417

2023 年 4 月 7 日

## 目录

<b>1</b>	算法介绍	
1.1	基于用户的协同过滤算法.....	2
1.2	基于物品的协同过滤算法.....	2
<b>2</b>	核心代码注解	
2.1	验证集划分.....	3
2.2	相似度计算.....	4
2.3	用户评分预测.....	7
2.4	预测结果可视化.....	8
<b>3</b>	结果分析	
3.1	基本公式.....	8
3.2	算法扩展相似度.....	10
3.3	退化成均值.....	10
3.4	UCF、ICF 结合.....	11
<b>4</b>	代码运行	
4.1	环境.....	12
4.2	运行.....	13
<b>5</b>	提交文件列表	
5.1	CODE.....	13

## 1 算法介绍

协同过滤 (Collaborative Filtering, CF) 的核心想法是利用某兴趣相投、拥有共同经验之群体的喜好来推荐用户感兴趣的信息。通过分析目标用户与其他用户的相似度、目标物品与其他物品的相似度, 过滤出相似的用户和物品, 通过他们的历史行为, 帮助预测目标用户对目标物品的喜爱程度。过滤不一定局限于特别相似的, 特别不相似信息的记录也相当重要。

协同过滤分为两种, 一种是基于用户的协同过滤 (User-Based Collaborative Filtering, UCF), 另一种是基于物品的协同过滤 (Item-Based Collaborative Filtering, ICF)

### 1.1 基于用户的协同过滤算法

UCF 是通过不同用户对同一件物品的评分来评测用户之间喜好的相似性, 基于用户之间的相似性做出推荐, 这种算法给用户推荐和他兴趣相投的其他用户喜欢的物品。

基于用户的协同过滤步骤如下:

1. 给定用户集合  $SU$ 、物品集合  $SI$ 、 $user - item$  评分矩阵  $mat [SU][SI]$ , 对  $SU$  中的每一对用户( $ua, ub$ ), 找到他们都评过分的共同物品, 加入集合  $Si$ ;
2. 考虑 $ua$ 、 $ub$ 的个人评分均值和对  $item \in Si$  的评分, 计算  $ua$ 、 $ub$  的相似度, 可采用计算等方法构造相似度矩阵  $sim [SU][SU]$ ;
3. 给定目标用户  $u$  和目标物品  $i$ , 对于  $k \in SU, k \neq u$ , 如果  $k$  评价过目标物品  $i$ , 那么将  $k$  加入  $u$  的邻居集合  $Sn$ ;
4. 对于  $n \in Sn$ , 用  $sim [u][n]$  和  $mat [n][i]$ , 结合  $u$  的个人评分习惯, 来预测  $mat [u][i]$ , 可采用平均值、加权平均值等方法。

### 1.2 基于物品的协同过滤算法

ICF 通过用户对不同物品的评分来评测物品之间的相似性, 基于物品之间的相似性做出推荐, 即, 给用户推荐和他之前喜欢的物品相似的物品。

基于物品的协同过滤步骤如下:

1. 给定用户集合  $SU$ 、物品集合  $SI$ 、 $user - item$  评分矩阵  $mat [SU][SI]$ , 对  $SI$  中的每一对物品 ( $ia, ib$ ), 找到用户  $user$ , 满足  $user$  给  $ia$ 、 $ib$  都打过分, 将这样的  $user$  加入集合  $Su$ ;
2. 对于  $user \in Su$ , 根据  $mat [user][ia]$  和  $mat [user][ib]$ , 计算  $ia$ 、 $ib$  的相似度, 可采用计算 Cosine Similarity (余弦相似度) 的方法, 构造相似度矩阵  $sim [SU][SU]$ ;
3. 给定目标用户  $u$  和目标物品  $i$ , 对于  $k \in SI, k \neq i$ , 如果  $u$  评价过  $k$ , 那么将  $k$  加入  $i$  的邻居集合 $Sn$ ;
- 4.对于  $n \in Sn$ , 用  $sim [i][n]$  和  $mat [u][n]$ , 来预测  $mat [u][i]$ , 可采用平均值、加权平均值等方法。

## 2 核心代码注解

本节解释代码的核心部分，所列函数非完整代码。主要代码语言为 Python，考虑到计算相似度的部分由于复杂度较高，用 Python 计算时间较长，不方便调试，后改用 C++ 编写。

### 2.1 验证集划分

函数定义: `split_valid_data(self, path, to_path, rate, round):`

参数含义:

`path`,: `train.csv` 路径

`to_path`: 划分好的验证集输出文件路径

`rate`: 验证集条目数占 `train.csv` 总条目数比例

`round`: 改变验证集挑选起点

引入用户活跃度和物品活跃度。用户活跃度定义为该用户评分过的物品总数，物品活跃度定义为给该物品评过分的用户总数。

为了保证训练数据的完整性，先读入 `train.csv`，构造 `user_item` 评分矩阵，并计算用户、物品活跃度。只挑选用户活跃度和物品活跃度均不低于 5 的用户-物品-评分条目，选入验证集，并从训练集中去除，更新用户、物品活跃度。

```
1      unum, inum = self.user_dict[i[0]], self.item_dict[i[1]]
      # 保证有足够的训练集
      if ( self.user_active[unum] >= 5 and self.item_active[inum] >= 5 ):
          # 写入验证集
          writer.writerow(i)
          # 从训练集中去除
          self.user_active[unum] -= 1
          self.item_active[inum] -= 1
          self.matrix[unum][inum] = 0
          cnt += 1
      if ( cnt == total ): # 收集满 total 条验证集
          break
```

由于下一步计算相似度转入C++，划分完训练集后需要将user-item 评分矩阵写入 .csv 文件中。

```
# 输出 ui_matrix
matwriter = csv.writer(open(matrix_path, 'w', newline=''))
for i in range (len(self.matrix)):
    matwriter.writerow(self.matrix[i])
```

## 2.2 相似度计算

由于相似度计算的复杂度为  $O(m^2n)$  (对于UCF,  $m$  为用户数量,  $n$  为物品数量; 对于 ICF,  $m$  为物品数量,  $n$  为用户数量), 试过用 Python 计算时间较长, 因此改成C++。

由于 UCF、ICF 的相似度计算过程类似, 只是循环的含义不同, 此处仅解释 UCF 的用户相似度计算过程。

首先需要读入划分完验证集之后在Python 中计算得到的user\_item\_matrix、用户打分平均值user\_ave、物品得分平均值item\_ave。

调用函数:

```
read_matrix(string path)
void read_list(string path, double * dst){
    ifstream inFile(path, ios::in);
    if ( ! inFile ) {
        printf("Open file failed. \n");
        return;
    }
    int i = 0;
    string line, field;
    while (getline(inFile, line)){// 按行读取 CSV 文件中的数据
        string field;
        istringstream sin(line); // 将整行字符串 line 读入到字符串流 sin 中
        getline(sin, field, ',');
        dst[i] = atof(field.c_str());
        ++ i;
    }
    inFile.close();
    cout << "total lines: " << i << endl;
    cout << "finish reading" << endl;
}
```

然后计算用户相似度(以余弦相似度为例)。

函数定义: user\_sim\_cal\_cosine(string to\_path)

参数含义: to\_path: 保存相似度矩阵的文件路径

```
void user_sim_cal_cosine(string to_path) {
    ofstream outFile(to_path, ios::out);
    if (!outFile) {
        printf("Open file failed. \n");
        return;
    }
    for (int ua = 0; ua < U; ++ua) {
        sim[ua][ua] = 1; // 自己和自己的相似度为 1
        for (int ub = ua + 1; ub < U; ++ub) {
            double numerator = 0, denominator_a = 0, denominator_b = 0;
```

```

    int simcnt = 0;
    for (int item = 0; item < I; ++item) {
        double r_ai = mat[ua][item];
        double r_bi = mat[ub][item];
        if (r_ai > 0 && r_bi > 0) { // a 和 b 同时给 i 打过分
            ++simcnt;
            numerator += r_ai * r_bi;
            denominator_a += r_ai * r_ai;
            denominator_b += r_bi * r_bi;
        }
    }
    if (simcnt > 0) {
        sim[ua][ub] = numerator / (sqrt(denominator_a) * sqrt(denominator_b));
        sim[ub][ua] = sim[ua][ub];
    }
    else {
        sim[ua][ub] = sim[ub][ua] = 0; // 没有共同打分
    }
}
for (int ub = 0; ub < U; ++ub) { // 写入一行
    outFile << sim[ua][ub];
    outFile << (ub == U - 1 ? '\n' : ',');
}
}
outFile.close();
cout << "finish writing similarity" << endl;
}

```

对与其他相似度计算公式也进行了尝试:

### Pearson 相似度

```

void user_sim_cal_pearson(string to_path){

    for ( int ua = 0; ua < U; ++ ua ){
        sim[ua][ua] = 1; // 自己和自己的相似度为 1
        for ( int ub = ua + 1; ub < U; ++ ub ){
            double r_a = user_ave[ua];
            double r_b = user_ave[ub];
            double E = 0, Sa = 0, Sb = 0;
            int simcnt = 0;
            for ( int item = 0; item < I; ++ item ){
                double r_ai = mat[ua][item];
                double r_bi = mat[ub][item];
                if ( r_ai > 0 && r_bi > 0 ){ // a 和 b 同时给 i 打过分
                    ++ simcnt;
                    E += ( r_ai - r_a ) * ( r_bi - r_b );
                    Sa += ( r_ai - r_a ) * ( r_ai - r_a );

```

```

        Sb += ( r_bi - r_b ) * ( r_bi - r_b );
    }
}
if ( simcnt > 0 && Sa * Sb != 0 ){
    sim[ua][ub] = E / sqrt(Sa * Sb);

}else{ // 没有共同打分或某一用户所有商品打分都一样

    sim[ua][ub] = 0;

}
sim[ub][ua] = sim[ua][ub];
}
}
}

```

### Improved Cosine Similarity (User-IIF)

```

void user_sim_cal_user_iif(string to_path) {

    vector<int> item_cnt(I, 0);
    for (int item = 0; item < I; ++item) {
        for (int user = 0; user < U; ++user) {
            if (mat[user][item] > 0) {
                ++item_cnt[item];
            }
        }
    }
    for (int ua = 0; ua < U; ++ua) {

        sim[ua][ua] = 1; // 自己和自己的相似度为1
        double u_a_iif = 0;
        for (int item = 0; item < I; ++item) {
            if (mat[ua][item] > 0) {
                u_a_iif += log(1 + U / item_cnt[item]);
            }
        }
        for (int ub = ua + 1; ub < U; ++ub) {
            double numerator = 0, denominator_a = 0, denominator_b = 0;
            int simcnt = 0;
            double u_b_iif = 0;
            for (int item = 0; item < I; ++item) {
                double r_ai = mat[ua][item];
                double r_bi = mat[ub][item];
                if (r_ai > 0 && r_bi > 0) { // a和b同时给i打过分
                    ++simcnt;
                    numerator += (r_ai - u_a_iif) * (r_bi - u_b_iif);
                    denominator_a += pow(r_ai - u_a_iif, 2);
                    denominator_b += pow(r_bi - u_b_iif, 2);
                }
            }
            if (simcnt > 0) {
                sim[ua][ub] = numerator / sqrt(denominator_a * denominator_b);
            } else {
                sim[ua][ub] = 0;
            }
        }
    }
}

```

```

    }
}
if (simcnt > 0) {
    denominator_a = sqrt(denominator_a);
    denominator_b = sqrt(denominator_b);
    double sim_value = numerator / (denominator_a * denominator_b);
    sim[ua][ub] = sim_value;
    sim[ub][ua] = sim_value;
}
else {
    sim[ua][ub] = sim[ub][ua] = 0; // 没有共同打分
}
}
}
}

```

### 2.3 用户评分预测

同样以 UCF 为例。按相似度由高到低遍历所有邻居，如果该邻居给目标物品打过分，那么记录相关数值。遍历完后，根据相应公式计算预测值。将预测出来的评分限制到【1, 5】的区间。

```

def predict_rating(self, unum, inum):
    """
    根据共现矩阵，计算一次 rating
    """
    r_u = self.user_ave_score[unum]
    total, weigh = 0, 0
    # i := neighbor index
    cnt = 0
    for u in range(len(self.user_dict)): # 遍历所有用户
        unum_b = int(self.neighbor[unum][u]) # 当前考察的邻居
        if ( unum == unum_b ):
            continue
        r_bi = self.matrix[unum_b][inum]
        if ( r_bi != 0 ): # 邻居给 inum 打过分
            r_b = self.user_ave_score[unum_b] # 该邻居打分均值
            total += ( r_bi - r_b ) * self.sim[unum][unum_b]
            weigh += self.sim[unum][unum_b]
            cnt += 1
        # 调整 top-K 邻居 #####
        if ( cnt == 1000 ): # 凑齐 cnt 个邻居
            break
    if ( weigh == 0 ):
        pred_ui = self.all_user_ave
        self.disable += 1
    else:
        pred_ui = r_u + total / weigh

```

```

if ( pred_ui < 1 ):
    pred_ui = 1
if ( pred_ui > 5 ):
    pred_ui = (self.all_user_ave+5)/2
return np.squeeze(pred_ui)

```

## 2.4 预测结果可视化

为了便于观察预测结果的分布情况了，在测试集上的拟合效果，将预测的结果可视化  
核心代码：

```

# 可视化预测结果
plt.style.use('ggplot')
plt.figure(figsize=(15, 4))
plt.scatter([i for i in range(len(predicted_list))], gt_rate_list, alpha=0.8, label='rate')
plt.scatter([i for i in range(len(predicted_list))], predicted_list, alpha=0.8,
label='predicted rate')
plt.legend(loc=[1, 1], fontsize=10)
plt.title('Prediction results on the validation set (RMSE: %.5f)'%RMSE)
plt.xlabel('index')
plt.ylabel('score')

```

## 3 结果分析

尝试了UCF、ICF 的不同公式、参数、组合。榜单上的提交大致如下：


尝试了User-IIF 、Pearson correlation、 Cosine Similarity 不同相似度进行计算


尝试了用Flyod 对相似度矩阵进行优化

因提交次数有限，未对本地所有预测结果进行提交

最好结果：Score: 0.95379

Rank	Username	Score	Time
4	Flyecnu	0.95379	6 1d





**Your Best Entry!**  
Your most recent submission scored 0.95379, which is an improvement of your previous score of 1.01344. Great job!

[Tweet this](#)

提交列表



✓	res_ucf.csv Complete · 1d ago	0.95379	<input type="checkbox"/>
✓	res_icf.csv Complete · 2d ago · self.user_ave_score[unum] 全是RMSE 0.9682152623924829	1.01344	<input type="checkbox"/>
✓	res_icf 0.8 quanshiall.csv Complete · 3d ago	1.11115	<input type="checkbox"/>
✓	res_uicf.csv Complete · 4d ago	1.04499	<input type="checkbox"/>
✓	res_icf.csv Complete · 5d ago	1.16789	<input type="checkbox"/>
⚠	res_icf.csv Error · 5d ago		
✓	res_ucf.csv Complete · 7d ago	1.0239	<input type="checkbox"/>

下面将预测分为4 部分作进一步的说明。

### 3.1 基本公式

前 1,2,6 次提交尝试了 UCF、ICF 及其基本的相似度计算公式。

对 UCF 来说，相似度计算采用Cosine Similarity 来计算。第 1 次使用Cosine Similarity 作为相似度，考虑最相似的 3 位邻居；第6 次使用余弦相似度，考虑所有邻居，按相似度加权平均，并进行Flyod 优化。

对ICF 来说，相似度计算可以采用余弦相似度（提交 2 ），本地尝试了课件上另有一个降低活跃用户权重的余弦相似度计算公式。

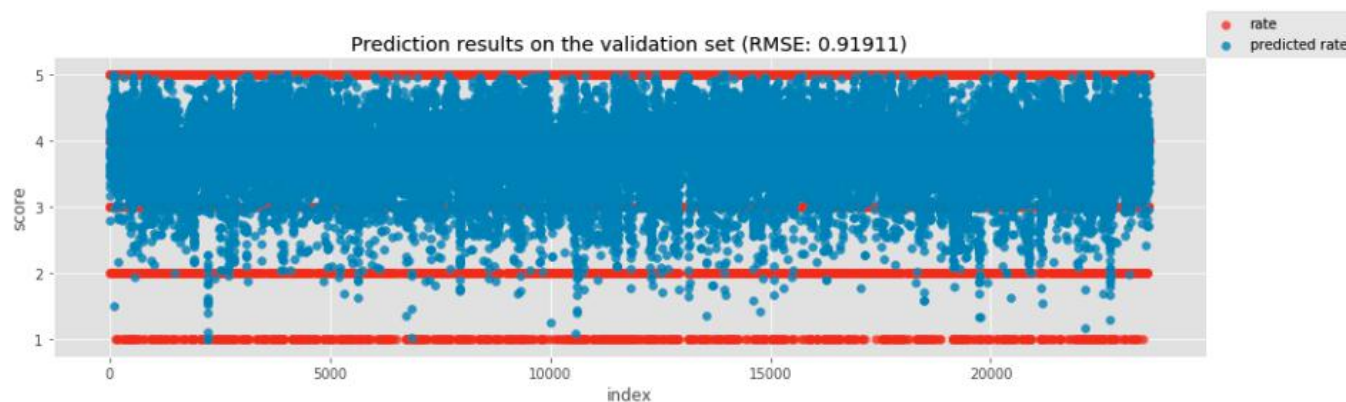


图1 第6 次提交在验证集上的预测结果

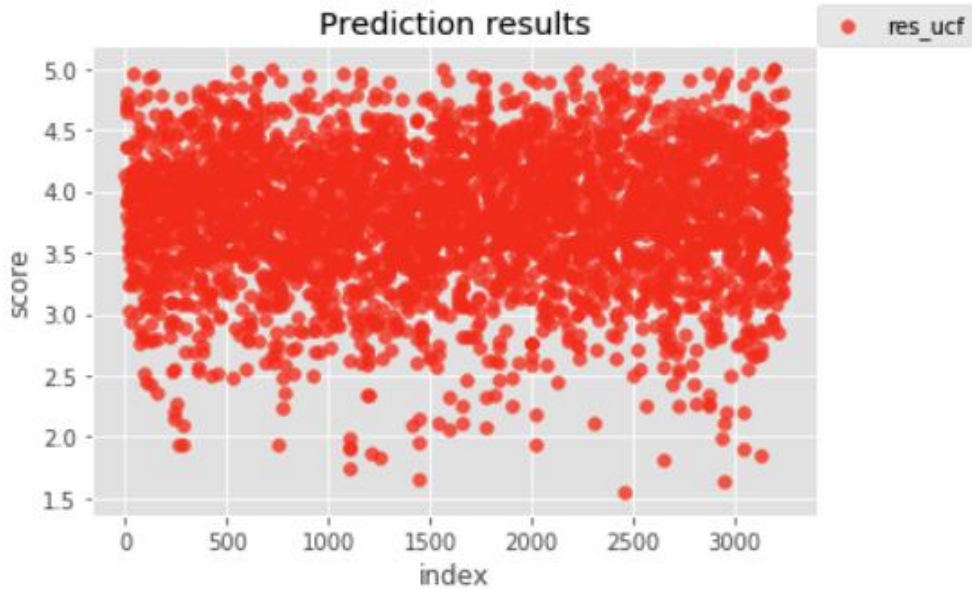


图2: 第6次提交在测试集上的预测结果

这 3 次提交总得来说差别较大, 预测点比较分散, 能涵盖 [1, 5] 的评分取值区间。同时预测错误带来的误差比较大。

### 3.2 Floyd 算法扩展相似度

考虑到数据集较小, 相似度矩阵十分稀疏, 想尝试能否按照“A 和 B 相似, B 和 C 相似, 那么 A 和 C 相似”以及“A 和 B 相似, B 和 C 不相似, 那么 A 和 C 不相似”的逻辑来填充相似度矩阵。

在 Floyd 最短路径算法的基础上修改, 用  $\text{sim}[A][C] * \text{sim}[C][B]$  来更新  $\text{sim}[A][B]$ 。

代码如下:

```
for ( int k = 0; k < U; ++ k )
    for ( int i = 0; i < U; ++ i )
        for ( int j = 0; j < U; ++ j )
            if ( sim[i][j] == 0 && sim[i][k] * sim[k][j] > 0 )
                sim[i][j] = sim[i][k] * sim[k][j];
```

使用这种方法确实能使得相似度矩阵不那么稀疏, 但是相似度相乘后得到的相似度往往值很小, 参考价值不大。但根据在kaggle上的测试结果, 这种方法对预测准确度有一定提高。

### 3.3 退化均值

由于相似度矩阵比较稀疏, 且尝试填充没有起到较好的效果, 只能考虑公式的退化。对于大部分用户、物品之间计算出的相似度是算不出来的, 找不到可以使用的邻居, 预测全部采用用户个人打分均值或所有用户打分均值。

另外，由于数据集太小，可以认为共同交互比较少的用户或物品之间算出来的相似度并不准确，不予考虑，仍旧退化成均值。

第4, 5 次提交中对共同交互的次数做了一个限制：对于两个物品，如果没有超过5 5 个用户同时给它们打过分那么就认为无法计算它们之间的相似度。

这一约束导致验证集测试集上只有个别点采用了预测，其他都由均值填充。

和图1 相比，相当于几乎没做预测，不符合数据集整体的分布规律， $RASE$  的值比较大。



图3:第5 次提交在验证集上的预测结果

### 3.4 UCF-ICF 结合

最后的第4 次提交尝试了综合上述提交，UCF 与 ICF 结合的方法。用户相似度的计算采用 Pearson correlation，物品相似度的计算采用余弦相似度，计算相似度时，共同交互至少为2，认为能够计算相似度。UCF 的预测将相似度阈值定在 0.1（考虑到基于用户的预测是在用户个人打分均值基准上浮动，相似度小的用户影响本身就比较小），ICF 的预测将相似度定在0.6。无法获得有效邻居的退化成所有用户打分均值。最后将 UCF 得到的预测值和 ICF 得到的预测值相加取平均。

结合 UCF、ICF 后，可以从图 4 看出，本地预测效果较好，但是实际打分结果不理想。

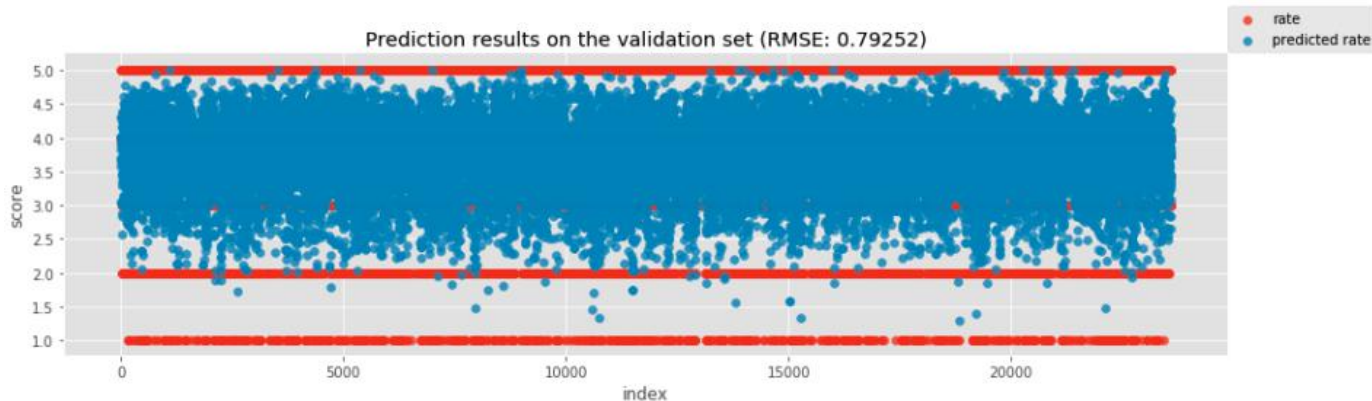
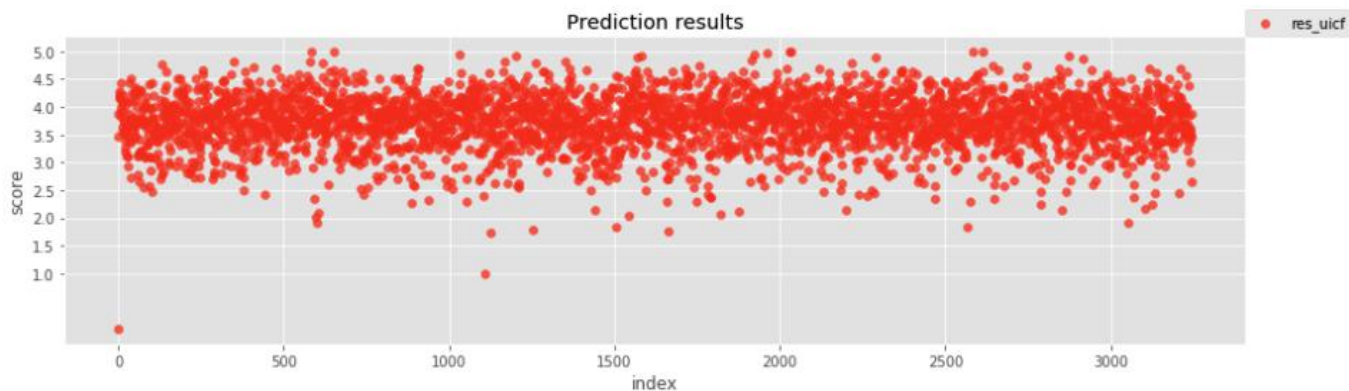


图4: 第4 次提交在验证集上的预测结果

图5:第 $\forall$ 次提交在测试集上的预测结果

## 4 代码运行

### 4.1 环境

PYthon 环境:

- Anaconda 3.5.2-Windows-x86-64
- Python 3.7

C++环境:

- gcc version 8.2.0

运行文件夹目录:

- Assign-1-data
  - train.csv
  - text.csv
- Assign-1-result
- usimcal.cpp
- isimcal.cpp
- ucf.ipynb
- iucf.ipynb
- icf.ipynb

## 4.2 运行

代码以.ipynb 和 .cpp 格式的文件提交。

**UCF** 运行方式:

依次执行 ucf.ipynb 中的 cell, 完成导入数据集、划分验证集、计算平均值等工作。

直到写有“# 计算 similarity C++”的 cell, 去运行 usimcal.cpp, 计算并导出用户相似度矩阵。

然后回到 ucf.ipynb, 继续执行 cell, 完成导入相似度矩阵、邻居排序, 以及在验证集上验证、在测试集上测试推荐效果, 并且可视化推荐结果。

**ICF** 运行方式:

依次执行 icf.ipynb 中的 cell, 完成导入数据集、划分验证集、计算平均值等工作。

直到写有“# 计算 similarity C++”的 cell, 去运行 isimcal.cpp, 计算并导出用户相似度矩阵。

然后回到 icf.ipynb, 继续执行 cell, 完成导入相似度矩阵、邻居排序, 以及在验证集上验证、在测试集上测试推荐效果, 并且可视化推荐结果。

**UICF** 运行方式:

依次执行 uicf.ipynb 中的 cell, 完成导入数据集、划分验证集、计算平均值等工作。

直到写有“# 计算 similarity C++”的 cell, 去运行 usimcal.cpp 和 isimcal.cpp, 计算并导出用户相似度矩阵。

然后回到 uicf.ipynb, 继续执行 cell, 完成导入相似度矩阵、邻居排序, 以及在验证集上验证、在测试集上测试推荐效果, 并且可视化推荐结果。

备注:

- 相似度矩阵文件导出、导入路径需匹配;
- simcal.cpp 中有多个计算相似度的函数可选, 代表不同的计算方法;
- 训练集不改变的话, 调整预测用的相似度阈值, 不需要重新计算相似度矩阵;
- 代码中涉及到修改预测所用阈值的部分已框出。

## 5 提交文件列表

### 5.1 Code

- ucf.ipynb: 基于用户的协同过滤算法代码
- icf.ipynb: 基于物品的协同过滤算法代码
- uicf.ipynb: 基于用户、物品混合的协同过滤算法代码
- usimcal.cpp: 计算用户相似度矩阵的代码
- isimcal.cpp: 计算物品相似度矩阵的代码

## 5.2 SimMatrix

- usim.csv: 用户相似度矩阵
- isim.csv: 物品相似度矩阵